Software Architecture

Abstract architecture was created with draw.io with basic relationships of the program being shown on the diagram.

Concrete architecture was produced jointly by PlantUML and Adobe Photoshop. The classes were separated into different categories with the connections within the category shown on the diagram. The inter-category connections are later added through Adobe Photoshop with lines colour coded for easier understanding.

Abstract Architecture

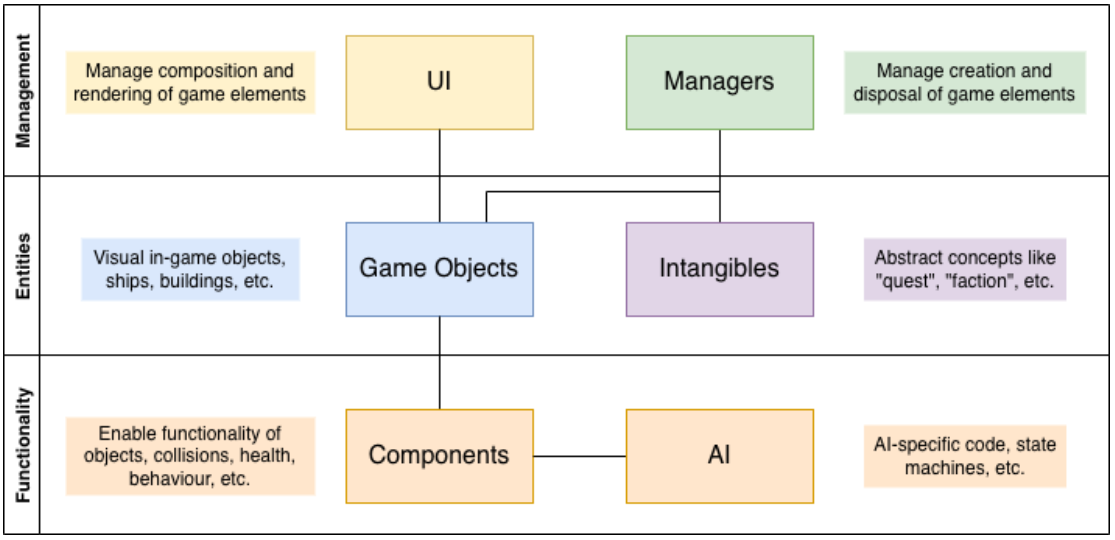

Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

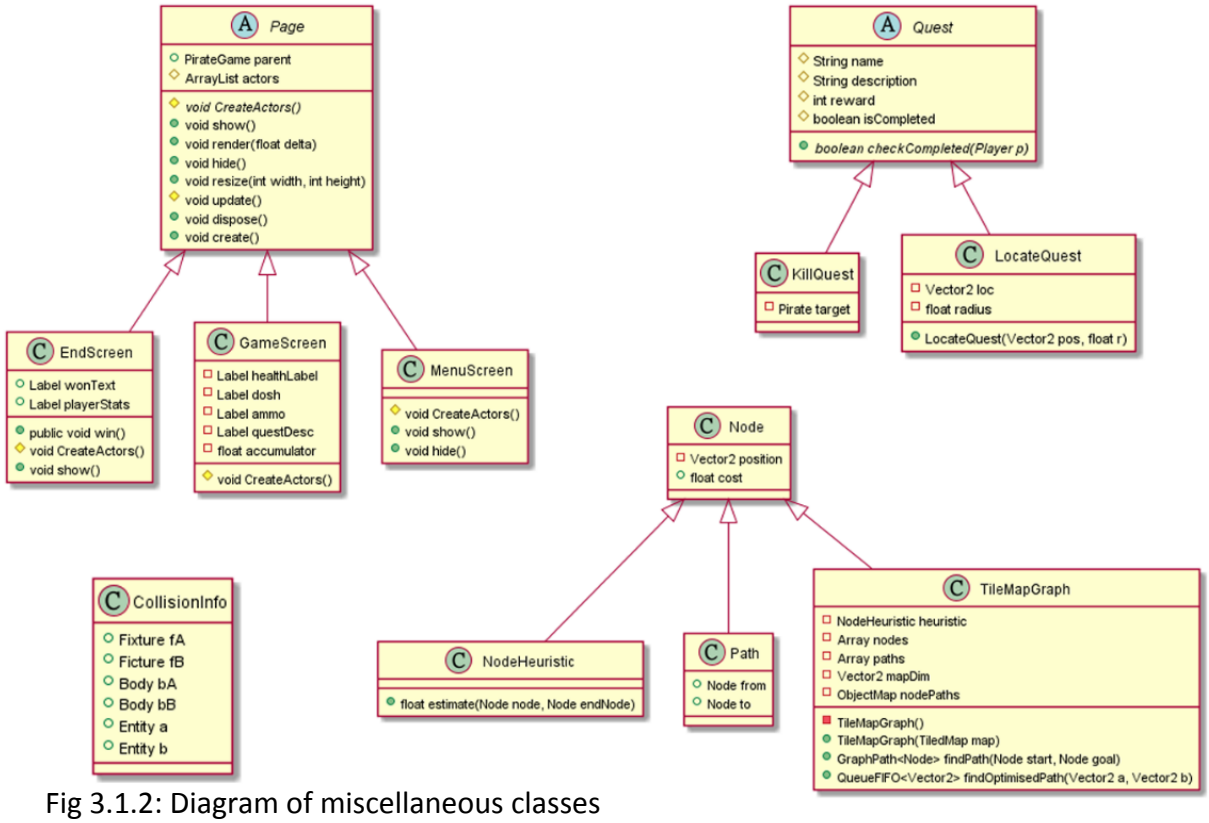

Fig 3.1.2: Diagram of miscellaneous classes

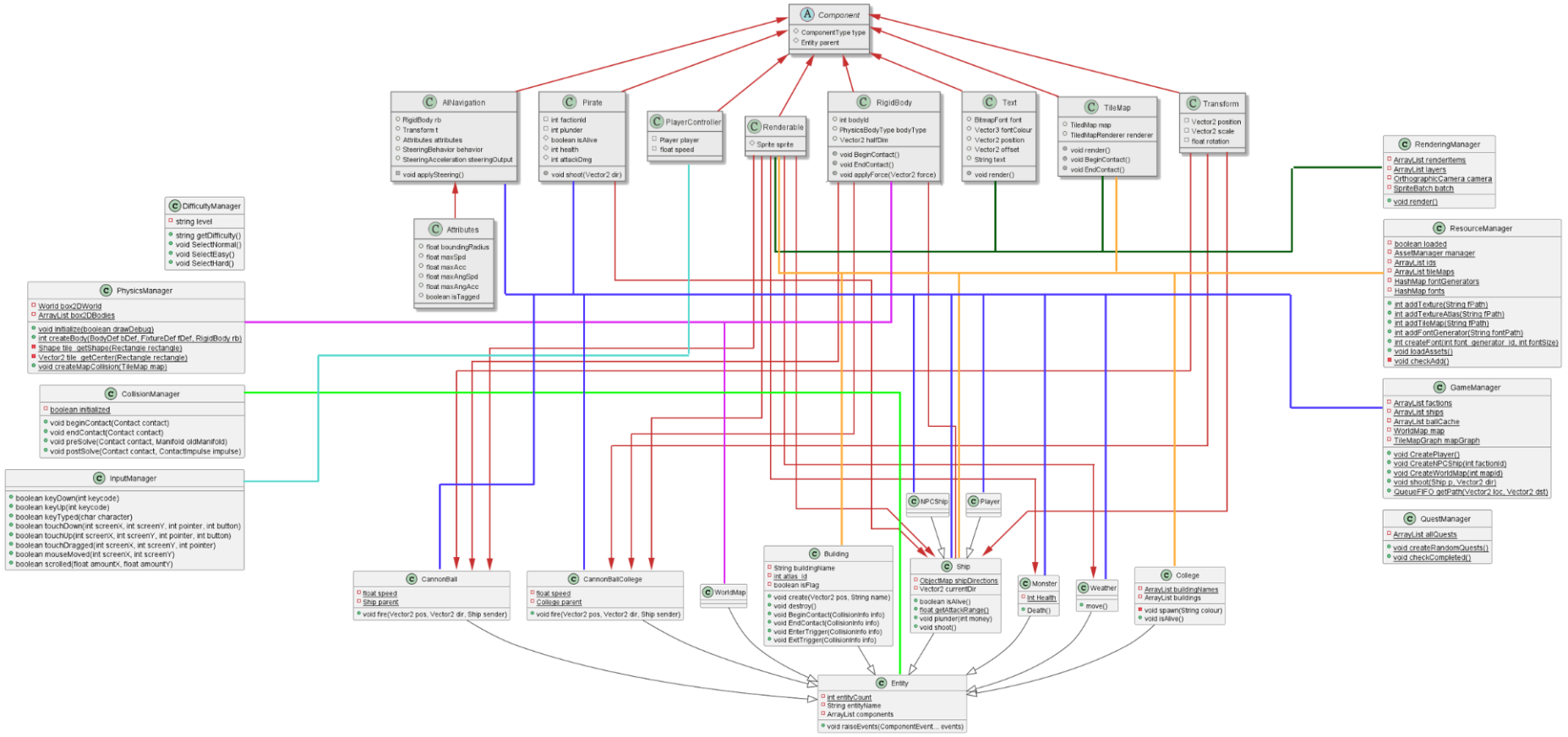

Fig 3.1.3: Diagram of Entity and Component classes

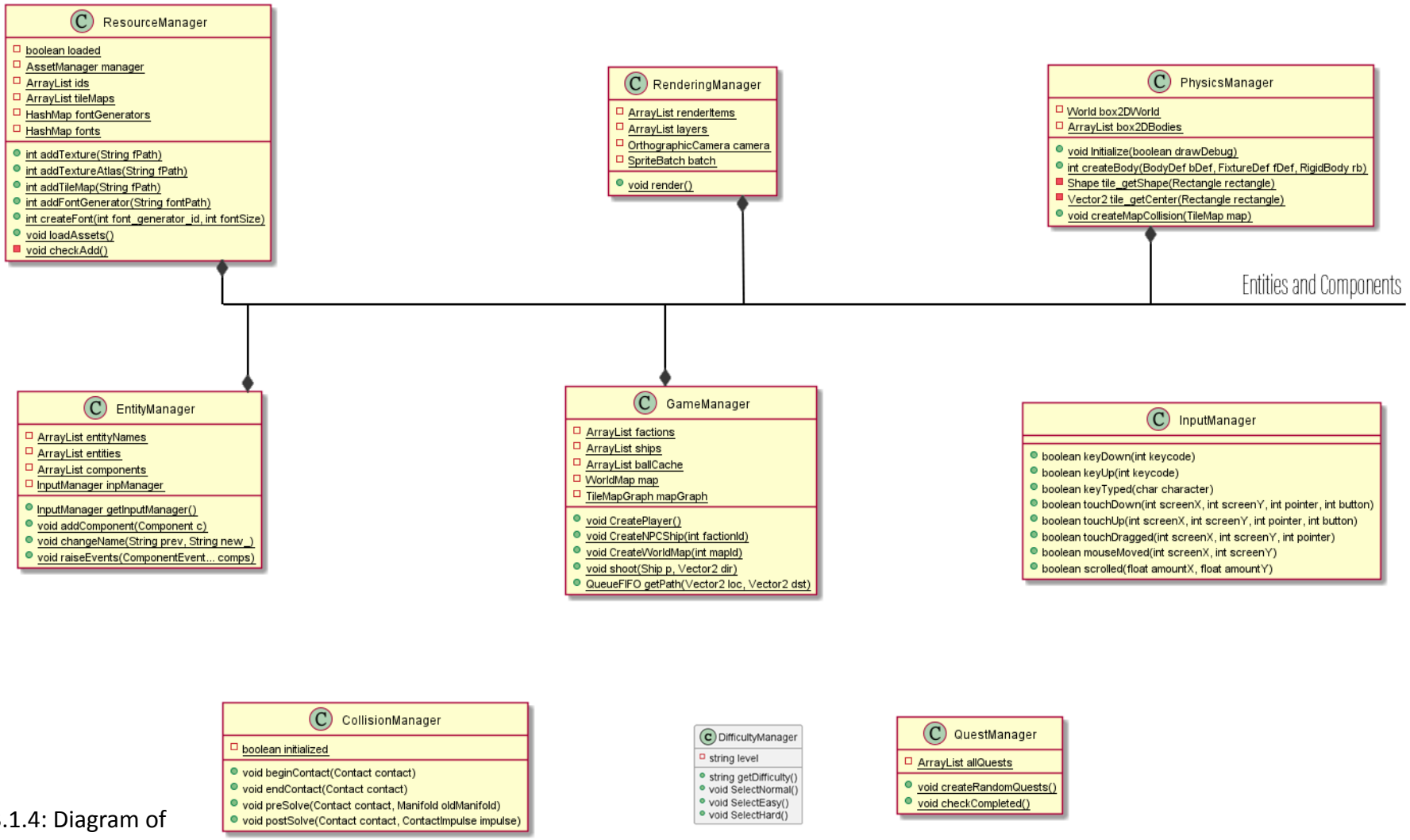

Fig 3.1.4: Diagram of Manager classes

The abstract architecture is concerned with segmenting the large, monolithic task of building the game into separate logical elements which could be planned and reasoned about separately. Connections drawn between elements signify a logical relationship rather than necessarily representing extension or composition relations such as those featured in the UML diagram detailing the concrete architecture. For example, factions/colleges ended up implemented as components and managed implicitly, unlike what fig. 3.1.1 seems to suggest. Nevertheless, it is useful to see them grouped under intangibles while planning the overall architecture.

Concrete architecture builds on the abstract in two main ways, by capturing additional implementation details, and by reflecting the contribution of the game engine to enabling game functionality.

Additional specifics of the game's implementation are provided by means of detailing the class structure of the code, annotating the classes with their significant functionality in the form of methods and variables, and drawing the relationships between the classes on the diagram.

The structure of the concrete architecture is informed by that of the game engine. For example, we move from the UI element of the abstract architecture to a separate Page class and its subclasses responsible for rendering and composition of UI widgets, and the Renderable component and RenderingManager class for the rendering of in-game objects such as ships and buildings: this is due to how the game engine implements the rendering of different game aspects. In this way, concrete architecture provides significantly more detail at a lower conceptual level than the abstract.

It should be noted that significant discretion had to be exercised regarding the level of detail captured in concrete architecture: it was neither feasible nor desirable to capture the full level of detail of the code's implementation. In the interest of using the concrete architecture as a higher-level abstraction used for reasoning about and planning the implementation, only significant functionality was captured and boilerplate methods and variables have been omitted. Furthermore, we had to deviate from the UML standard to depict certain relationships without making the diagrams too large to display on A4 paper. Hence, figs 3.1.3 & 3.1.4 have the relationships between entities & components and their respective managers depicted in a shorthand form that we hope is nevertheless clear and informative.

Another point of note regarding the architecture and implementation is that during the process of implementation, certain approaches were selected that were not obvious during the architecture planning stage. For example, update methods called by the game loop were leveraged to provide certain functionality, like monitoring for game over conditions within the GameScreen class. These approaches were not foreplanned and are hard to document within a UML class diagram. Hence, a better reference to them would be perusing the rendered Javadocs associated with the game.

Relations to requirements

FR_SHIP_KB_INPUT
By referring on the InputManager in fig 3.1.4, there are numerous functions in it which accepts keyboard signals from the user for ship navigation

FR_VIEWPORT_SCALING
By referring to the Page class on fig 3.1.2, there is a class resize() which takes the width and height of the display or window, thus being able to render the game on displays with different sizes.

FR_PLAYER_FIRE
Referring to the Ship class in fig 3.1.3, there is a function called shoot which is called the same function in the GameManager in fig 3.1.4, which allows users to fire weapons.

FR_BULLET_TRAVEL
Referring to CanonBall class in fig 3.1.3, the method fire() takes the starting position, direction and the sender ship, thus it shows the travel of the munitions sent from ships.

FR_QUEST_TRACKING / FR_QUEST_RANDOMISE
In the Quest class of fig 3.1.2 and Quest manager of fig 3.1.4, there are methods called checkCompleted() and createRandomQuests(), which showed the game can track on player's quest completions and also randomise quest objectives.

FR_GAME_WIN
In EndScreen class on fig 3.1.2, there is a label called wonText and a method called win(), which are responsible for displaying status of the completion of boss encounter.

NFR_WORLD_COLLISIONS
There are multiple classes and methods which are responsible for world collisions. In PhysicsManager(fig 3.1.4), there is a method createMapCollision() which is responsible for creating zones which can be collided into. In both Building and TileMap classes (fig 3.1.3), there are methods called BeginContact() and EndContact() which process the collision of entities in the game. And data of two entities colliding will be stored in class Collisioninfo of fig 3.1.2.